DRAWING BOARD

Designer RAM

ROBERT GROSSBLATT,
CIRCUITS EDITOR

EVER SINCE CP/M BEGAN TO DECLINE, people have been saying that the days of the Z80 were numbered—don't you believe it. It's true that you won't see many new computer designs done around a Z80, but it's also true that the Z80 has too much muscle to wind up on the silicon scrap heap. It's still one of the microprocessors of choice to use as a dedicated controller.

Building our dynamic-RAM system around a Z80 makes sense because the chip's built-in features relieve us of the burden of implementing much of the design in external hardware. We'll still need glue to put all the pieces together, but not anywhere as much if we were building the whole circuit from gates alone.

There are four Z80 control signals that are critically important in the construction of our circuit. Understanding what they are, how they work, and what their timing relationships are, is the first step in the design. The four signals, all of which are active low, are:

- Memory Request ($\overline{MREQ}$), pin 19
- Read ($\overline{RD}$), pin 21
- Write ($\overline{WR}$), pin 22
- Refresh ($\overline{RFSH}$), pin 28

Let's discuss them one at a time.

$\overline{MREQ}$ is a control signal that is active whenever the Z80 has been instructed to perform an operation that involves external memory. As soon as the Z80 has an address ready to put out on the bus it brings this line low. That happens for *all* memory operations: read, write, and refresh.

$\overline{RD}$ goes low when the Z80 wants data from the outside world, which can be from either a memory location or an I/O port. Therefore a request for a read from memory must be sensed by watching $\overline{MREQ}$ as well as $\overline{RD}$.

$\overline{WR}$ is the opposite of $\overline{RD}$. When it goes low, the Z80 has data that it wants to send to either memory or an I/O device. Just as with $\overline{RD}$, the destination is determined by watching the $\overline{MREQ}$ line.

$\overline{RFSH}$ is the signal that keeps the Z80 popular. When it goes low it signals that the microprocessor has incremented its internal refresh counter and has put the new refresh address on the lower seven bits of the address bus (A0–A6). By combining $\overline{RFSH}$ signal with $\overline{MREQ}$, you can determine exactly when a refresh operation must take place in your system.

All memory operations require two Z80 control signals, so it's important that we have a good understanding of the timing relationships between them. And any discussion of timing must start with a look at the basic heartbeat of the Z80: the instruction cycle.

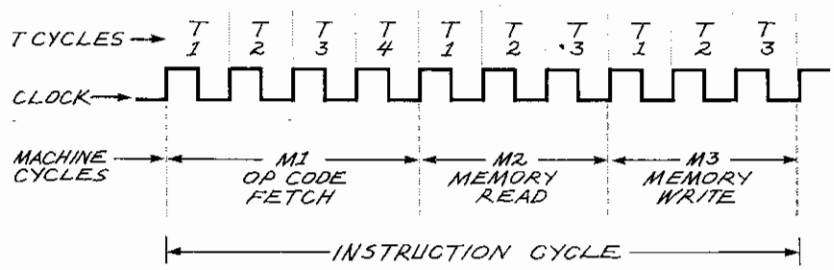

FIG. 1

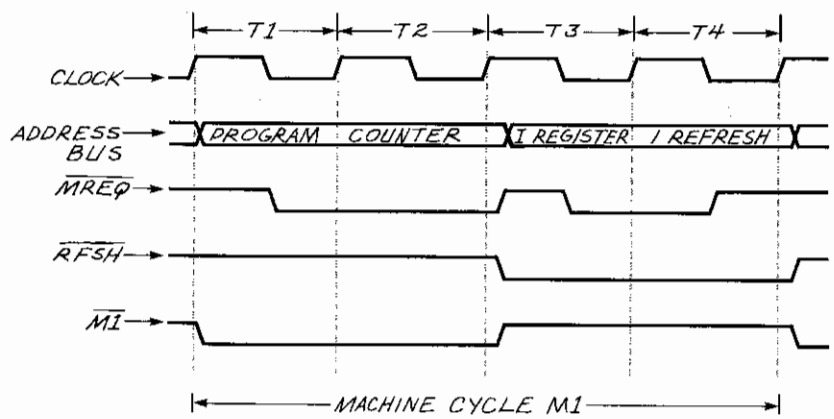

FIG. 2

M and T cycles

Figure 1 is a representation of the two fundamental parts of all Z80 instruction cycles: the M (machine) cycle, and the T (time, or clock) cycle. Every instruction that

the Z80 executes requires from one to six machine cycles, M1–M6. During M1 the Z80 fetches the op-code of the next instruction. If the op-code is more than one byte long there will be more than one M1 cycle. In addition, it's during M1 that the Z80 handles refresh addressing. By the way, as shown in the figure, M2 and M3 are used for reads and writes.

Figure 2 is an expanded look at the M1 cycle. During T1 and T2, the Z80 places the contents of the program counter on the the address bus to get the next op-code. The microprocessor uses the next two T cycles, T3 and T4, to decode the op-code; it doesn't need the bus during that time. So, during T3 and T4, the address bus is divided in half to provide two kinds of data. The upper eight bits, A7–A15, have the contents of the I (index) register, and the lower seven bits, A0–A6, have the contents of the R, or refresh, register.

When the refresh address stabilizes, both $\overline{MREQ}$ and $\overline{RFSH}$ go low. That combination of signals is therefore a guaranteed-stable refresh address that can be used to systematically refresh dynamic memory.

In case you missed it, what the Z80 is doing for us is to eliminate the need for the external counters and logical glue that used to be necessary, to ensure that dynamic memory would be refreshed at the right time and in the right order.

Now that you understand how much work the Z80 is ready to do for us, let's see what we have to do to take advantage of it.

Putting it to work

In using dynamic RAM with a Z80, the most important design task is to ensure that the memory is fast enough to work in the amount of time available for refresh. In our circuit we'll use a Z80B and run it at a maximum speed of 2.5 MHz, which translates into 400 nanoseconds ($1/(2.5 \times 10^6)$) per T cycle, or 800 ns to complete one refresh. In fact, however, we can't count on having the full 800 nanoseconds. Some time is eaten by delays internal to the Z80; more is needed to allow for propagation delays in our support circuitry; and all dynamic RAM needs a refresh "precharge" time. Keeping that in mind, let's see how much of the 800 nanoseconds we actually have for refresh.

Even though Fig. 2 is only a representation of actual timing, it's clear that there are delays associated with the memory-control signals generated by the Z80. It takes about three quarters of a T cycle for the Z80 to put the program counter on the address bus and then guarantee that the memory control signals ($\overline{MREQ}$ and $\overline{RD}$) are stable. And after the op-code appears on the data bus, we also must allow for settling time on the data bus.

Assuming the maximum clock speed of 2.5 MHz, we can expect to see the following timing for the whole op-code fetch cycle, (T1–T2):

$$T(\text{OP-CODE}) = (T1+T2) - T(\text{ADDRESS/SIGNAL}) - T(\text{DATA})$$
$$= 800 - 300 - 50$$
$$= 450 \text{ nanoseconds}$$

The amount of time needed for a memory read is also an important consideration, but it is usually longer than the time needed for an op-code fetch. For our 2.5 MHz system, a memory read requires about 650 nanoseconds. Not only is that longer than an op-code fetch, but both numbers are well within the bounds of the modern 150-nanosecond (and faster) RAM.

The logic that we'll need to make our system work also has built-in delays. Each of the buffers and gates that comprise the circuit contributes to the total amount of time the circuit needs to operate. TTL and fast CMOS parts have very small propagation delays, but if you add enough of them together you can wind up with a circuit that is too slow. A worst-case analysis might look like this:

- 40 ns for memory buffer delays
- 40 ns for data buffer delays
- 30 ns for gating delays
- 40 ns for Z80 buffer delays

That's a total circuit-propagation delay of 150 ns.

Those figures are not exact, but if you look through a TTL or CMOS data book, you'll see that I've overestimated the maximum possible times by a large margin.

Now that we have all the numbers worked out, we also have the maximum access time for the RAM

we'll need. You don't need a lot of equipment to see that even if we used some slow 250-nanosecond RAM, the total circuit delay time would only be 400 (150 plus 250) ns. Most mail-order houses don't even stock 250-ns parts. The bottom line is that by using 150-ns RAM we can eliminate all the potential problems that would be caused by timing restrictions.

System design

There are a couple of things to keep in mind when using a Z80 to control a RAM system. All of them come from one general principle: *If the Z80 stops running, all our memory data will be history.*

That principle is of critical importance and it's also really easy to forget. As long as the Z80 is executing instructions it will continue putting out new refresh addresses during each M1 cycle. However, anything that puts the Z80 to sleep will also trash the memory. Fortunately, there are only a few circumstances in which that can happen:

- A reset pulse longer than 1 millisecond.
- A wait state longer than 1 millisecond.
- A DMA operation longer than 1 millisecond.

In our system, all memory access will be done through the Z80, we don't have to worry about the last two on the list. DMA simply won't be used in our system; any external request to store or retrieve data will be done by loading latches and then asking the Z80 to perform a read or a write. Similarly, we simply don't have any wait states.

Any slow I/O device using our memory system will talk to buffers and latches, not directly to memory. Some memory systems (like that of the IBM PC) must place wait states into every memory request because there isn't enough time for the "precharge time" required by the dynamic RAM. As we've seen from the mathematical analysis above, we surely don't have that problem.

Next month we'll start building the circuit. If you haven't done so already, you should get good data sheets on the Z80 and dynamic RAM. **R-E**